Exam-March-2021-Solutions

Alejandro Russo

March 22, 2021

Contents

1 Exam/Ex1.hs

```haskell
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}

module Exam.Ex1
where

import GHC.TypeLits

import Control.Applicative
import Prelude hiding (Monoid)

import Data.Proxy


-- Define a monoid based on a carrier t
data Monoid t = Empty | Elem t | Plus (Monoid t) (Monoid t)

-- 5pt: Define a graded monad only with static information
data GMonad (o :: Monoid t) m a = GMonad { unGMonad :: m a }

-- 5pt: Define return and bind
returnGMonad :: Monad m => a -> GMonad Empty m a
returnGMonad a = GMonad (return a)

bindGMonad :: Monad m => GMonad e1 m a -> (a -> GMonad e2 m b) -> GMonad (Plus
    e1 e2) m b
bindGMonad (GMonad m) f = GMonad $ m >>= unGMonad . f

runGMonad :: GMonad e m a -> m a
runGMonad (GMonad m) = m

-- 15pt: Proof that a graded monad is a monad
{-
Assumptions:

unGMonad . GMonad = id
GMonad . unGMonad = id


Law 1:
return x >>= f =?= f x

  return x >>= f
== def. return
  GMonad (return x) >>= f
== def. bind
  bindGMonad (GMonad (return x)) f
== def. bindGMonad
  GMonad $ return x >>= unGMonad . f
```

```
== Law 1. underlying monad
  GMonad (unGMonad (f x))
== def. (.)
  (GMonad . unGMonad) (f x)
== by GMonad . unGMonad = id
  f x

Law 2:
m >>= return =?= m

I know, by definition that, m = GMonad m' for some m'

   m >>= return
== def. bind
   bindGMonad (GMonad m') return = GMonad $ m' >>= unGMonad . return
== def. return in GMonad
   bindGMonad (GMonad m') return = GMonad $ m' >>= unGMonad . GMonad . return
== unGMonad . GMonad = id by definition and (.) is associative
   bindGMonad (GMonad m') return = GMonad $ m' >>= return
== Law 2 of underlying monad
   bindGMonad (GMonad m') return = GMonad $ m'
== by assumption m = GMonad m'
   m

Law 3:
(m >>= f) >>= g =?= m >>= (\x -> f x >>= g)

Assumption m = GMonad m'

   (m >>= f) >>= g
== def. bind.
   bindGMonad (bindGMonad (GMonad m') f) g
== def. bindGMonad
   bindGMonad (GMonad $ m' >>= unGMonad . f) g
== def. bindGMonad
   GMonad $ (m' >>= unGMonad . f) >>= unGMonad . g
== Law 3 of underlying monad
   GMonad $ m' >>= (\x -> (unGMonad . f) x >>= unGMonad . g)
== def. (.)
   GMonad $ m' >>= (\x -> unGMonad (f x) >>= unGMonad . g)
== by Aux. Lemma
   GMonad $ m' >>= (\x -> unGMonad . (f x >>= g))
== By scope variable x
   GMonad $ m' >>= unGMonad . (\x -> f x >>= g))
== def. bind
   m >>= (\x -> f x >>= g)

Auxiliary lemma:
unGMonad m >>= unGMonad . f =?= unGMonad (m >>= f)

   unGMonad (m >>= f)
== def. bind
   unGMonad (bindGMonad m f)
== def. bindGMonad where m = GMonad m' & m' = unGMonad m by definition
   unGMonad (GMonad $ unGMonad m >>= unGMonad . f)
== def. (.)
   (unGMonad . GMonad) $ unGMonad m >>= unGMonad . f
```

```
== unGMonad . GMonad = id
     unGMonad m >>= unGMonad . f

-}
```

2 Exam/Ex2.hs

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE RebindableSyntax #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

module Exam.Ex2 where

import Prelude hiding ((>>), (>>=), return, Monoid)
import GHC.TypeLits
import Exam.Ex1
import Data.Proxy

----------------------------------------

--Rebindable syntax
return = returnGMonad
(>>=)  = bindGMonad
(>>)   = \x y -> x >>= const y

----------------------------------------

-- Define a resource that it is counted
sWriteFile :: FilePath -> String -> GMonad (Elem 1) IO ()
sWriteFile file str = GMonad $ writeFile file str

----------------------------------------

-- Introduce assertions about the effects, and that will require the type family
-- to "normalize".

type family Norm (e :: Monoid Nat) :: Nat where
  Norm (Empty)      = 0
  Norm (Plus e1 e2) = Add (Norm e1) (Norm e2)
  Norm (Elem t)     = t


type family Add (e1 :: Nat) (e2 :: Nat) :: Nat where
  Add x y = x + y

assertMaxWrites :: (Norm e ~ w, w <= max)
                => Proxy max -> GMonad e m a -> GMonad (Elem w) m a
assertMaxWrites p (GMonad m) = GMonad m

-- A computation that performs two writes
twoWrites = do
  sWriteFile "file1" "hello"
  sWriteFile "file2" "bye"

fourWrites = twoWrites >> twoWrites

-- This examples should type-check correctly
```

```haskell
okWrites = assertMaxWrites (Proxy :: Proxy 2) twoWrites

-- These examples should not type-check due to failed assertions
-- badWrites  = assertMaxWrites (Proxy :: Proxy 1) twoWrites

----------------------------------------
--Control more than one effect, for instance, read and writes into files

type family Norm2 (e :: Monoid (Nat, Nat)) :: (Nat, Nat) where
  Norm2 (Empty)      = '(0, 0)
  Norm2 (Plus e1 e2) = Add2 (Norm2 e1) (Norm2 e2)
  Norm2 (Elem t)     = t

type family Add2 (e1 :: (Nat, Nat)) (e2 :: (Nat, Nat)) :: (Nat, Nat) where
  Add2 '(x1, y1) '(x2, y2) = '(x1 + x2, y1 + y2)

sReadFile' :: FilePath -> GMonad (Elem '(1, 0)) IO String
sReadFile' file = GMonad $ readFile file

sWriteFile' :: FilePath -> String -> GMonad (Elem '(0, 1)) IO ()
sWriteFile' file str = GMonad $ writeFile file str

assertMaxWrites' :: (Norm2 e ~ '(r, w), w <= max)
                 => Proxy max -> GMonad e m a -> GMonad (Elem '(r, w)) m a
assertMaxWrites' p (GMonad m) = GMonad m

assertMaxReads' :: (Norm2 e ~ '(r, w), r <= max)
                => Proxy max -> GMonad e m a -> GMonad (Elem '(r, w)) m a
assertMaxReads' p (GMonad m) = GMonad m

-- A computation that performs two writes and one read

twoWritesOneRead = do
  sWriteFile' "file1" "hello"
  s <- sReadFile' "file2"
  sWriteFile' "file3" s

-- This examples should type-check correctly

good =  assertMaxWrites' (Proxy :: Proxy 2)
      $ assertMaxReads'  (Proxy :: Proxy 2)
      $ twoWritesOneRead

-- These examples should not type-check due to failed assertions
{-
notgood1 = assertMaxWrites' (Proxy :: Proxy 2)
         $ assertMaxReads'  (Proxy :: Proxy 0)
         $ twoWritesOneRead

notgood2 = assertMaxWrites' (Proxy :: Proxy 1)
         $ assertMaxReads'  (Proxy :: Proxy 2)
         $ twoWritesOneRead
-}
```